Secure Software Engineering: A Knowledge Modeling based Approach for Inferring Association between Source Code and Design Artifacts

Chaman Wijesiriwardana[1]
Faculty of Information Technology
University of Moratuwa
Katubedda, Sri Lanka

Ashanthi Abeyratne[2], Chamal Samarage[3],
Buddika Dahanayake[4], Prasad Wimalaratne[5]
University of Colombo School of Computing
Reid Avenue, Colombo 07, Sri Lanka

***Abstract*—Secure software engineering has emerged in recent decades by encouraging the idea of software security has to be an integral part of all the phases of the software development lifecycle. As a result, each phase of the lifecycle is associated with security-specific best practices such as threat modeling and static code analysis. It was observed that various artifacts (i.e., security requirements, architectural flaws, bug reports, security test cases) generated as a result of security best practices tend to be segregated. This creates a significant barrier to resolve the security issues at the implementation phase since most of them are originated in the design phase. In order to address this issue, this paper presents a knowledge-modeling based approach to semantically infer the associations between architectural level security flaws and code-level security bugs, which is manually tedious. Threat modeling and static analysis are used to identify security flaws and security bugs, respectively. The case study based experimental results revealed that the architectural level security flaws have a significant impact on originating security bugs in the code level. Besides, the evaluation results confirmed the scalability of the proposed approach to large-scale industrial software products.**

***Keywords*—Software security; threat modeling; knowledge modeling; security flaws**

I. Introduction

Having identified the critical need for software security, the paradigm shift of *"Building Security In"* has emerged in the recent decades [1], [2], [3]. This paradigm shift requires software security to be addressed in all phases of the software development lifecycle. Literature reveals that most security vulnerabilities result from defects that are unintentionally introduced in the software during the design phase and the implementation phase [2]. Garry McGraw has identified code reviews and architectural risk analysis as the top two best practices to minimize the security vulnerabilities in software systems [2]. These best practices are called as *security touchpoints* associated with the artifacts produced by the implementation phase (i.e., codebase) and the design phase (i.e., design documents) respectively. Even in organizations with mature software development processes, the artifacts created are segregated from each other [4]. Furthermore, to the best of our knowledge, existing tools are not capable of identifying security-specific associations between the artifacts generated during software development. This reveals a significant research gap of interlinking the artifacts originated at the implementation phase and the design phase.

This paper presents a conceptual framework and a proof-of-concept implementation to semantically interlink architectural level security flaws and code-level security bugs based on the foundation laid in [5]. Security flaws are identified based on STRIDE [6], [7] threat categorization model introduced by Microsoft, which helps to identify threats from the attackers' perspective by classifying attackers goals into six threat categories. Security bugs are determined based on OWASP Top 10 [8], [9], [10] vulnerabilities, which are the ten most critical web application security risks providing a great awareness for web application security. In this paper, security flaws and bugs are interlinked by employing a knowledge-modeling based technique, which facilitates inferring the associations that are manually tedious.

In this approach, rather than directly interlinking different artifacts, it is required to infer the associations among design documents and source code to reveal whether the root causes for security bugs lie in the design phase. Knowledge-modeling based approaches are capable of handling large quantities of data intelligently [11] and proven successful in the domain of cybersecurity [12]. Besides, expertise in software security is not readily available. Therefore, knowledge-modeling approaches are useful when security expertise is not available. It also provides a common platform for integrating knowledge on a large scale. Finally, knowledge bases are capable of generating new knowledge by using the stored data.

More precisely, the key contributions of this paper are:

- a knowledge model and the association rules to infer the hidden relationships across security flaws and bugs, and
- results of the evaluation experiments conducted by using the proof-of-concept implementation.

The remainder of this paper is organized as follows: Section 2 presents the related work. Section 3 describes the proposed approach to interlink security artifacts. Section 4 presents the proof-of-concept implementation followed by the evaluation in Section 5. Section 6 concludes and present future work.

II. Background and Related Work

Building Security In[2] is a collaborative effort that provides practices, tools, guidelines, rules, principles, and other resources that software developers, architects, and security practitioners can use to build security into software in every phase of its development. Correspondingly Microsoft has carried out a noteworthy effort under its Trustworthy Computing Initiative which focused on people, process, and technology to tackle the software security problem [13]. On the people front, Microsoft trains every developer, tester, and program manager in basic techniques of building secure products. Microsoft's development process has been enhanced to make security a critical factor in design, coding, and testing of every product. A key part of Microsoft's Trustworthy Computing is the Security Development Lifecycle (SDL) which focuses on software development and introduces security and privacy throughout all phases of the software development process. The Microsoft SDL combines a holistic and practical approach to reduce the number and severity of vulnerabilities in Microsoft products [1]. Conforming to the aforementioned approaches introduced to the SDLC, it conveys that Architectural risk analysis and Code review are two significant steps which should be conducted in a security specific SDLC process.

A. Architectural Risk Analysis

Frydman et al. [14] introduce an automated approach for threat modeling by producing two data structures: identification trees and mitigation trees. Identification trees are used to determine threats in the software design, while mitigation trees describe countermeasures of threats by classifying software specifications that are required to mitigate a specific risk. The two data structures and ranking information of threats have combined in a knowledge base called *attack patterns*. Yuan et al. [15] describe their approach to develop a tool to retrieve relevant common attack pattern enumeration and classification (CAPEC) type attack patterns for software development. CAPEC attack patterns are valuable resources that can help software developers to think like an attacker and have the potential to be used in each phase of the secure software development lifecycle. A metric has been defined in this tool to measure the degree of usefulness of an attack pattern and the degree of its relevance to a particular STRIDE category. Berger et al. [16] propose a practical approach to architectural risk analysis that leverages Microsoft threat modeling approach. This proposed approach uses extended DFDs and a security knowledge base to aid software developers in detecting vulnerabilities in software architectures. The knowledge base contains information on architectural weaknesses and possible mitigations. However, these approaches explicitly operate only on the design phase with no effort to interlink the threats with source code level bugs.

B. Security Specific Code Analysis

A practical approach for implementing secure practices into the software development lifecycle outlined in [17]. It has introduced a development testing platform which allows the development organizations to coherently integrate code testing into the software development process. Coverity development testing solutions train developers to address both security and quality when testing the code which leads to secure software development practices. The commonly found potentially critical security defects in the source code are identified from this platform, which is an aid for the developers to fix them. The major weakness of this platform is that it mainly focuses on implementation phase without considering the actual root-causes lie in the design phase. Alqahtani et al. [18] have stressed the fact that despite the security concerns are reported in specialized vulnerability databases; these repositories often remain information silos. As a solution, they introduced a modeling approach that eliminates these silos by linking security knowledge with other software artifacts to improve traceability and trust in software products. A Security Vulnerabilities Analysis Framework (SV-AF) is introduced in this approach to support evidence-based vulnerability detection. This approach also explicitly focus the code level and connecting the design phase with the identified bugs is not addressed.

1) Interlinking static analysis with other artifacts: Interlining of software artifacts has been extensively discussed in the literature [19], [20], [21]. Implementation vulnerabilities differentiate themselves from the design vulnerabilities because they only exist in the source code and are not part of the original design or requirements. The implementation vulnerabilities are also very language-specific, especially the C and C++ coding languages are infamous for their ease of creating implementation vulnerabilities [22], [23]. The languages memory control is both its strength and weakness. The control the developer has creates the opportunity to create optimized and fast software but also insecure code that can easily be exploited. Some of the most common causes of implementation vulnerabilities are buffer overflows, format string bugs, integer overflows, null dereferences, and race conditions.

A novel approach for implementing secure practices into the software development lifecycle outlined in [17]. It has introduced a development testing platform which allows the development organizations to coherently integrate code testing into the software development process. Coverity development testing solutions train developers to address both security and quality when testing the code which leads to secure software development practices. The commonly found potentially critical security defects in the source code are identified from this platform, which is an aid for the developers to fix them. The major weakness of this platform is that it mainly focuses on implementation phase without considering the actual root-causes lie in the design phase. Alqahtani et al. [18] have stressed the fact that despite the security concerns are reported in specialized vulnerability databases; these repositories often remain information silos. As a solution, they introduced a modeling approach that eliminates these silos by linking security knowledge with other software artifacts to improve traceability and trust in software products. A Security Vulnerabilities Analysis Framework (SV-AF) is introduced in this approach to support evidence-based vulnerability detection. This approach also explicitly focus the code level and connecting the design phase with the identified bugs is not addressed.

In contrast, our approach is not limited to ensuring the security in a single phase of the software development lifecycle or a single artifact originated from a specific lifecycle phase. Instead, we attempt to semantically interlink the artifacts produced in the design phase and implementation phase. It allows software practitioners to identify the root-causes for the security bugs.

III. Approach

This approach aims at inferring the associations between security flaws and security bugs introduced during the design phase and the implementation phase of the lifecycle. As stated previously, security flaws are identified in terms of STRIDE threat categorization, and security bugs are represented regarding OWASP top 10 vulnerabilities. The approach consists of three main constituents: *threat modeling* to identify security flaws, *static code analysis* to identify security bugs, and exploiting *knowledge base* to infer relationships among flaws and bugs. The conceptual architecture of the proposed approach, exhibiting its external data sources, internal components, boundaries, and their associations, is depicted in Fig. 1.

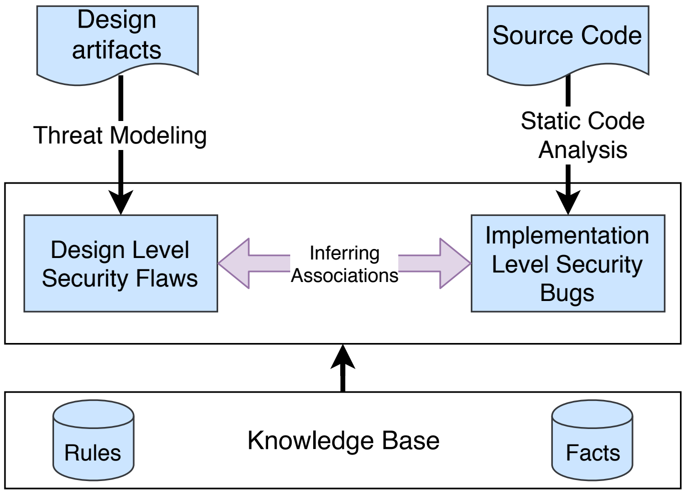

Fig. 1. Conceptual Model to Infer Associations between Flaws and Bugs.

A. Identifying and Pre-Processing Security Flaws

Security flaws are identified through an architectural risk analysis, which includes explicitly identifying security risks in the software architecture/design. In this paper, threat modeling used as the architectural risk analysis method due to several noteworthy reasons such as the ability to work with high-level design diagrams, simplicity to employ in different contexts, and explicit tool support. For example, threat modeling process can be initiated by drawing a data flow diagram (DFD). According to Abi-Antoun et al. [24], architectural level security flaws can effectively identify by analyzing Level 0 or Level 1 DFDs. As depicted in Fig. 2, threat modeling process consists of thee steps.

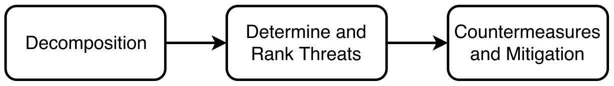

Fig. 2. Main Steps of the Threat Modeling Approach.

1) Decomposition:: This step concerned with gaining an understanding of the application and how it interacts with external entities. This knowledge helps in identifying entry points to see where a potential attacker could interact with the application, determining trust levels which represent the access rights that the application will grant to external entities and identify assets that the attacker would be interested. Thus, this information is used to produce data flow diagrams (DFDs) for the application.

2) Determine and rank threats:: In this step, threats are determined and categorized according to a threat categorization methodology. The goal of threat categorization is to identify threats from both attackers perspective and defensive perspective. DFDs produced in step 1 is primarily used to identify potential threat targets from the attackers perspective.

3) Countermeasures and mitigation:: In this step, mitigations, and countermeasures are identified for the ranked threats.

B. Using Static Analysis to Identify Security Bugs

Static analysis is used to detect the security bugs that appear in the source code of the software project. For adequately understanding the code level security bugs, they are categorized based on OWASP Top 10 vulnerabilities. OWASP Top 10 is the ten most critical web application security risks which provide a powerful awareness document for web application security. The different versions of OWASP T10 are focused on identifying the most common vulnerabilities which depict how an attacker can potentially harm a software system. Though the static analysis detects the bugs that are categorized into OWASP vulnerabilities, that information is not sufficient to generate a relationship with security flaws. Therefore, it was decided to utilize OWASP Proactive Controls[1], which is a set of developer-centric security techniques that can be included in the every software project. Most importantly, each proactive control helps in preventing one or more of the OWASP Top Ten web application security vulnerabilities. OWASP top 10 vulnerabilities have been mapped to proactive controls.

C. Inferring Relationships among Flaws and Bugs

Inferring logical relationships between security flaws and bugs are expected to obtain via STRIDE and OWASP. However, STRIDE mainly focuses on attacking perspective of software security. On the other hand, Application Security Frame (ASF)[2] is a threat categorization model, which helps to identify the threats from the defensive perspective. For an in-depth analysis of the threats affecting the software application data and functional assets, both the STRIDE attacker view and the ASF defensive view for the enumeration of threats considered as essential. As stated previously, threat modeling process only focuses on attacker's perspective based on STRIDE. Therefore, to involve the defensive standpoint, a relationship has been identified between STRIDE and ASF.

What the relationship depicts by UcedaVelez and Morana [25] is not a complete association between ASF and STRIDE due to each category of STRIDE lacks an association to ASF type. Hence, this association further improved by combining the findings of Adam Shostack[6]. Table I presents the improved mapping between STRIDE and ASF.

[1]https://www.owasp.org
[2]https://msdn.microsoft.com/en-us/library/ff649461.aspx

TABLE I. MAPPING BETWEEN STRIDE AND ASF

STRIDE Attack Type	ASF type
Spoofing	Authentication
Tampering Information Disclosure Elevation of privileges	Authorization
Repudiation Elevation of privileges	Configuration Management
Tampering Information Disclosure	Data Protection in Storage and Transit
Tampering	Data Validation / Parameter Validation
Information Disclosure	Error Handling and Exception Management

1) Semantic Text Similarity between ASF and Proactive Controls: The semantic text similarity calculated for every single security control in ASF with every single Proactive Control. The descriptions of ASF security controls and Proactive Controls are not limited to a single phrase. Accordingly, the semantic text similarity of each phrase of the description of a particular ASF security control calculated concerning each phrase of the description of Proactive Control. Consequently, the average semantic similarity score between a specific ASF security control (Ai) and Proactive control (Pi) calculated as follows.

$$SemS(Ai, Pi) = \left[\left(\sum_{i=0}^{nm} Vi \right) \div nm \right] \tag{1}$$

where:

A_i = description of ASF having n phrases
P_i = description of proactive controls having m phrases
V_i = similarity between ASF and proactive control

2) Knowledge Base: Security specific information about software projects can be in in the form of either structured or unstructured in heterogeneous information sources. Some of these information sources may frequently undergo significant changes as well. Thus, a knowledge modeling approach would more practical and beneficial in developing a security framework for software development. The knowledge base of this approach contains the facts and rules related to the STRIDE, ASF, OWASP T10, Proactive Controls and Semantic Similarity Scores between ASF and Proactive controls. A Frame-based approach is used for knowledge representation of facts [26]. The structure of the frames for the facts STRIDE, OWASP T10, and Similarity Matching are as follows.

Listing 1: Frame for STRIDE Categories

```
frame (stride,
[category_model [val threat],
types - [val [spoofing, tampering,
information disclosure,
denial of service, elevation of privileges]]
])
```

Listing 2: Frame for OWASP Categories

```
frame (owasp_t10,
[category_model [val bug],
types - [val [a1, a2, a3, a4, a5,
              a6, a7, a8, a9, a10]]
```

Listing 3: Frame for Semantic Similarity

```
frame(semantic_similarity
[proactive_control [val c1],
security_control [val s1],
score [val value]
```

Prolog rules were designed to infer the association between STRIDE and OWASP T10.

Rule 1: Querying the Knowledge-base

```
isCausedByThreatCategories
(BugCategory, TList_Unique) :-
findall(T, isCausedByThreatCategory
(BugCategory, T), TList),
sort(TList, TList_Unique)
```

Rule 1 is used to query the knowledge base. The list of unique threat categories can be discovered by querying the knowledge base using a bug category. Each threat category associated with bug category is revealed by the Rule 2.

Rule 2: Discovering the associated threat category using the bug category

```
isCausedByThreatCategory(BugCategory, T) :-
lacksProactive(BugCategory, P),
mapsToSecurityControl(P, S),
isWeakendByThreatCategory(S, T)
```

Rule 2 is used to discover the associated threat category using the bug category. The threat category is revealed using the subsequent rules on the right-hand side of Rule 2. The lacksProactive(BugCateogry, ProactiveControl) is used to discover the proactive controls violated due to the given bug category.

Rule 3 - Identifying the proactive controls of the relevant bug categories

```
lacksProactive(BugCategory, C) :-
isProactiveListOf(CList, BugCategory),
member(C, CList)
```

Rule 3 is used to identify the proactive controls of the relevant bug categories in succession.

Rule 4 - Discovering the associated threat category using the bug category

```
isProactiveListOf(CList, BugCategory) :-
owasp_top10(BugCategory, _ , CList)
```

The Rule 4, isProactiveListOf(ProactiveControlList, BugCategory) used to identify the proactive list of the given bug category using the owasp_top10 frame.

Rule 5 - Discovering the associated threat category using the bug category

```
mapsToSecurityControl(Proactive, S) :-
isMappingSecurityControlList(SList,Proactive),
member(S, SList)
```

Rule 5 is used to identify the mapping ASF security controls in succession by using the semantic text similarity score between ASF security controls and proactive controls. An ASF security control is mapping with a proactive control if it belongs to the top three semantic text similarity scores of the relevant proactive control. Additional six rules are used to identify the mapping security controls using the semantic text similarity scores, which is not listed here due to space limitations.

Rule 6 - Discovering the associated threat category using the bug category

```
isWeakendByThreatCategory(SecurityControl,T):-
stride(_ , T , _ , SecContList),
member(SecurityControl, SecContList)
```

The association results given by the Knowledge-base are used to create the associations between Bugs and Threats. The facts regarding STRIDE and ASF security controls are static facts while OWASP T10 and Proactive Controls are dynamic facts. The reason for keeping OWASP T10 and Proactive Controls as dynamic is that both of these facts are continuously getting revised based on technological advancements and industrial best practices. Thus, the knowledge base generates new knowledge based on the regular updates.

IV. Proof-of-Concept Implementation

In this research, as a proof-of-concept, a security analysis framework has implemented to infer the relationships between design flaw and implementation bugs by adhering to the theoretical foundation described in the previous section. Figure 3 provides an overview of the proof-of-concept implementation and its main constituents: (a) STRIDE Categorization of Security Flaws, (b) OWASP Categorization of Security Bugs, and (c) Knowledge-base and Association inferencing.

A. STRIDE Categorization of Security Flaws

As described previously, design-level security flaws are identified by using Threat Modeling. A threat model will be generated by analyzing the Level-0 or Level-1 DFD of the software system under investigation. The threat model is created using the Microsoft Threat Modeling Tool (TMT), which categorizes the threats according to the STRIDE model. The threat model generated from Microsoft TMT obtained as an XML file, which is further processed to extract the *threats*. After that, the extracted threats converted into *threat objects* that contains the relevant details of the threats introduced based on the proposed design as depicted in the DFD. Then *threat category objects* are created based on STRIDE categorization, which facilitates identifying the specific threat category of a specific threat introduced in the design phase. Users can either upload an external DFD or manually draw it.

B. OWASP Categorization of Security Bugs

SA-SEC facilitates code analysis in two distinct ways. First, it can use SonarQube to analyze the source code. SonarQube allows the categorization of the vulnerabilities identified as security bugs into OWASP Top 10. However, *SA-SEC* does not entirely depend on a third-party tool like Sonarqube for bug categorization. A novel mechanism based on Case-based reasoning [27] is introduced to categorize the bugs into OWASP categories by analyzing the different attributes of identified vulnerabilities. A collection of attributes such as *threat description, threat type, etc* that are common across multiple security tools have identified. Table II provides a list of the selected attributes together with a short description.

TABLE II. Attributes Used for Case-based Reasoning

Attribute	**Description**
Threat	The description of the vulnerability where it includes a clause related to actual problem. Examples: “Code should not be dynamically injected and executed”, “Credentials should not be hard-coded”
Type	It can be a Bug, a Vulnerability or a Code Smell
Severity	Five severity levels are considered. **BLOCKER**: Bug with a high probability to impact the behavior of the application in production. The code must be immediately fixed. **CRITICAL**: A bug with a low probability to impact the behavior of the application. The code must be immediately reviewed. **MAJOR**: Quality flaw which can highly impact the developer productivity. **MINOR**: Quality flaw which can slightly impact the developer productivity. **INFO**: Neither a bug nor a quality flaw, just a finding.
Effort	The time (in minutes) estimate to fix the issue and update the tests.
Technical Debt	The time estimate to fix all maintainability issues (Code smells).
Language	The programming language that the issue occurs. For example : Javascript, Java, C#

Upon identifying and categorizing the security bugs, they are converted into *bug objects*. Each *bug object* contains the relevant details of bugs that will be the output of this component. The *bug objects* sent out by the Security Bug Pre-processor are transformed into Bug category objects. Ten Bug category objects are created with respect to OWASP T10 which contains details of each Bug object belongs to a particular category. Theses Bug category objects are sent to the Association Loader component.

C. Knowledge base and Association Inference Module

Association Loader used for querying the Knowledge Base. A Prolog converter is developed using SWI-Prolog to communicate with Java. Each bug category will be used to query the Knowledge Base, and the associated threat type

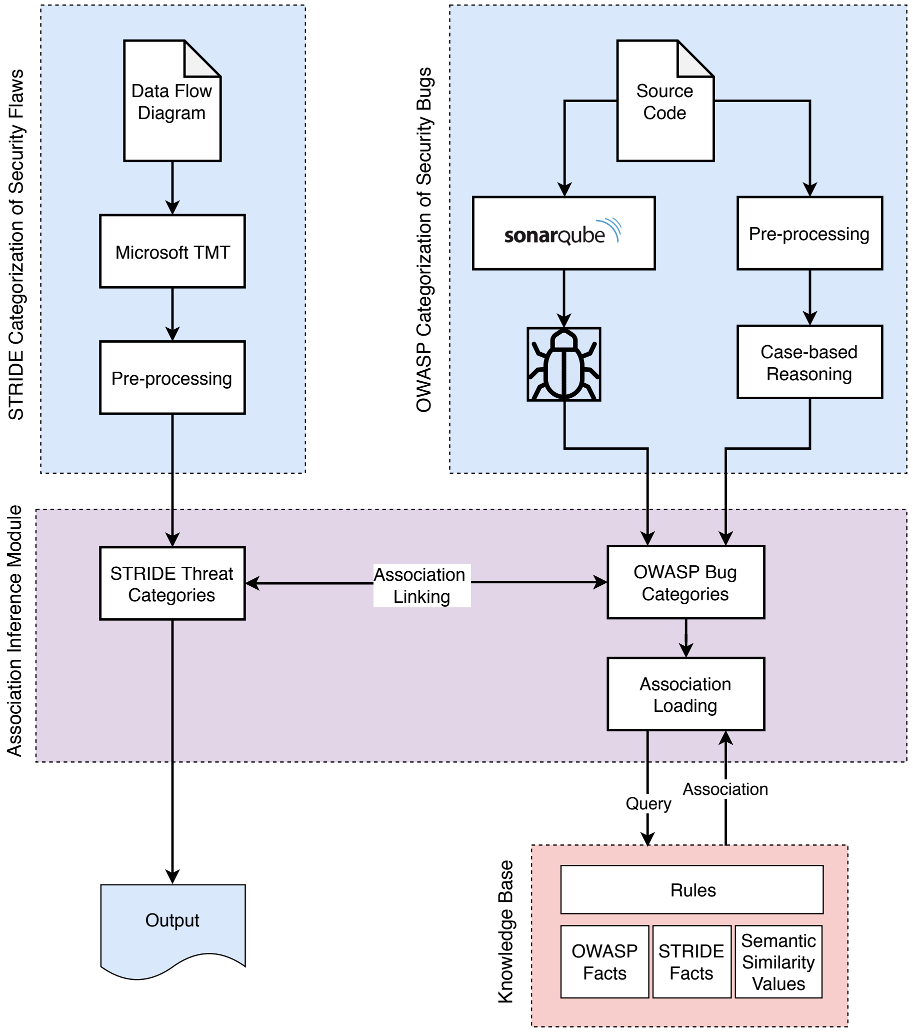

Fig. 3. Architecture of the Proof-of-Concept Implementation.

results held inside the Association Loader. The associated threat type results and the Bug Category Objects are sent to the Association Linker. Threat category objects from STRIDE Transformer and associated threat types and Bug objects from Association Loader will be the input to the association linker. After that, the Association objects generated. The Association objects are sent out from the Association Linker to the Output Builder.

The Knowledge Base is built using the SWI-Prolog. All the facts and rules described previously are contained in the Knowledge Base. The Knowledge Base has the capability of updating when the OWASP categories or Proactive controls revised. On the other hand, knowledge base explicitly allows expanding the knowledge contained in it using the additional knowledge of security experts.

V. Evaluation

The main focus of the evaluation is to find whether the potential root causes of an identified security bug lie in the design phase of the software application. Two case studies have employed for the evaluation process.

a) Case study 1 - User Authentication component: : Fig. 4 presents the DFD of the user authentication in a web-based application. It consists of two processes, one external entity, and a single data store together with associated data flows. In the evaluation process, threat modeling is conducted to identify the architectural-level security flaws, and static analysis is used to capture the security bugs at the implementation level. The association derived between security bugs and the threats are based on the security bug categories and threat categories. Table III depicts the possible threats identified by the threat modeling process. Similarly, through static code analysis, A2, A5, and A6 categories[3] of OWASP have captured. The results produced from the threat modeling process and the static code analysis provided as input to find the association between them. The derived associations present in Table IV.

[3]A1 to A10 are the top 10 threat categories of OWASP

Then, the highly relevant causes of the security bug categories were identified, and the corresponding countermeasures were applied to remove the security bugs in the source code. After repeating static analysis, it was observed that previously designated A2, A5 and A6 bug categories were removed successfully. Hence, it was evident that by removing the potential security specific root causes at the design level leads to resolve the security bugs at the code level.

TABLE III. IDENTIFIED THREATS OF THE USER AUTHENTICATION COMPONENT

Threat Type	No: of Threats
S	6
T	4
R	4
I	2
D	9
E	8

TABLE IV. ASSOCIATIONS DERIVED FOR CASE STUDY 1

OWASP Type	Derived association
A2	S, T, R, I, E
A5	T, R, I, E
A6	S, T, R

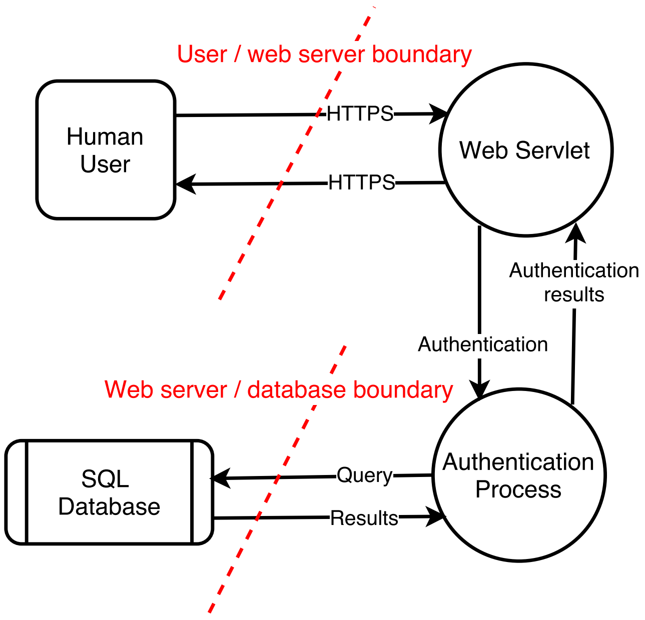

Fig. 4. DFD of the user Authentication Component of a Web Application.

b) Case study 2 - Large-scale web based application:
: A large-scale industry project is selected to evaluate the scalability of the proposed approach. Fig. 5 presents a part of the DFD of the application. The full diagram is not presented due to its complexity with higher number of processes and

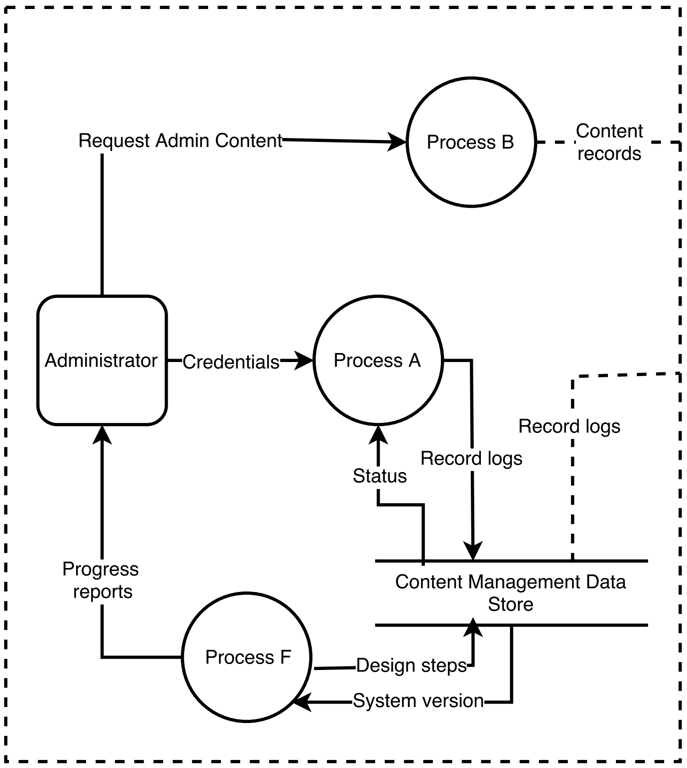

Fig. 5. Part of the DFD of the Large-Scale Web based Application.

data stores. Similar to Case study 1, DFD diagram is subjected to threat modeling and code base subjected to static analysis using the tools mentioned above. Summary of the identified threats presents in Table V. Based on the static code analysis, 26 security bugs related to A2 and A6 categories of OWASP have identified. The results produced from the threat modeling process and the static code analysis provided as input to find the association between them. The associations between threats and bugs that are derived from this approach is presented in Table VI. Then the corresponding countermeasures were applied to remove the security bugs in the source code. After repeating static analysis, it was observed that previously designated A2, A6 bug categories were removed successfully.

TABLE V. IDENTIFIED THREATS OF THE LARGE-SCALE INDUSTRY APPLICATION

Threat Type	No: of Threats
S	12
T	0
R	0
I	5
D	5
E	5

A. *Threats to Validity*

The accuracy of the results obtained from the experiments depends on the analysis outputs given by SonarQube and MS Threat Modeling Tool. Despite the fact that the associations derived from this approach depict the possible causes for a security bug, even with such an association, pinpointing the

TABLE VI. Associations Derived for Case Study 2

OWASP Type	Derived association
A2	S, T, R, I, E
A6	S, T, R

exact location of the source code is improbable with a Level 0 DFD. Therefore, it is essential to consider the lower level DFDs at the threat modeling phase for efficient capturing of security bugs.

On the other hand, static code analysis tools may not be capable of capturing all the bug categories in OWASP Top 10. Hence, this approach is unable to derive associations for each OWASP Top 10 vulnerabilities contained in the source code. Therefore a manual code review is required to identify the remaining vulnerabilities. However, manual code reviews are not feasible with large-scale, complex projects.

VI. Conclusions

This paper presents a Knowledge-modeling approach to infer the associations among design artifacts and source code to reveal whether the root causes for security bugs lie in the design phase. This research employed a frame-based approach for the knowledge representation of security-specific information extracted from design documents and source code. Evaluation results imply that the knowledge-modeling approach successfully detects whether design flaws are propagated to the implementation phase. Besides, this paper provides experimental evidence of the usefulness and applicability of the concept of *Building Security In*. Moreover, this research contributes to the body of knowledge in secure software engineering by filling the research gap in interlinking security artifacts.

This approach has several limitations. First, security vulnerabilities at the design phase are detected solely based on DFDs, which is mainly due to the unavailability of tools to discover vulnerabilities of the other types of design artifacts such as UML diagrams. Secondly, the proof-of-concept implementation is entirely depending on the results produced by the Threat Modeling Tools and Static Analysis Tools. Thus, the derived associations are also could be biased to those tools.

The framework could serve as a stepping stone to the researchers in the field of software security, which was lacking previously. On the other hand, the knowledge base has provisions to evolve with different security aspects. As future work, the experiments are expected to repeat for large-scale open-source software systems. Furthermore, it is planned to improve this research to directly interlink security bugs with security flows by utilizing attack trees or case-based reasoning.

Acknowledgment

The authors gratefully acknowledge the financial support provided by the National Research Council of Sri Lanka (Grant no: NRC 15-74).

References

[1] S. Lipner, "The trustworthy computing security development lifecycle," in *Computer Security Applications Conference, 2004. 20th Annual*. IEEE, 2004, pp. 2–13.

[2] G. McGraw, *Software security: building security in*. Addison-Wesley Professional, 2006, vol. 1.

[3] M. Kreitz, "Security by design in software engineering," *ACM SIGSOFT Software Engineering Notes*, vol. 44, no. 3, pp. 23–23, 2019.

[4] G. Antoniol, G. Canfora, G. Casazza, and A. De Lucia, "Information retrieval models for recovering traceability links between code and documentation," in *Software Maintenance, 2000. Proceedings. International Conference on*. IEEE, 2000, pp. 40–49.

[5] A. Abeyratne, C. Samarage, B. Dahanayake, C. Wijesiriwardana, and P. Wimalaratne, "A security specific knowledge modelling approach for secure software engineering," *Journal of the National Science Foundation of Sri Lanka*, vol. 48, no. 1, 2020.

[6] A. Shostack, *Threat modeling: Designing for security*. John Wiley & Sons, 2014.

[7] R. Khan, K. McLaughlin, D. Laverty, and S. Sezer, "Stride-based threat modeling for cyber-physical systems," in *Innovative Smart Grid Technologies Conference Europe (ISGT-Europe), 2017 IEEE PES*. IEEE, 2017, pp. 1–6.

[8] D. Wichers and J. Williams, "Owasp top-10 2017," *OWASP Foundation*, 2017.

[9] S. M. Srinivasan and R. S. Sangwan, "Web app security: A comparison and categorization of testing frameworks," *IEEE Software*, vol. 34, no. 1, pp. 99–102, 2017.

[10] M. Willberg, "Web application security testing with owasp top 10 framework," 2019.

[11] W. E. Zhang and Q. Z. Sheng, *Managing Data From Knowledge Bases: Querying and Extraction*. Springer, 2018.

[12] Y. Jia, Y. Qi, H. Shang, R. Jiang, and A. Li, "A practical approach to constructing a knowledge graph for cybersecurity," *Engineering*, vol. 4, no. 1, pp. 53–60, 2018.

[13] M. Howard and S. Lipner, *The security development lifecycle*. Microsoft Press Redmond, 2006, vol. 8.

[14] M. Frydman, G. Ruiz, E. Heymann, E. César, and B. P. Miller, "Automating risk analysis of software design models," *The Scientific World Journal*, vol. 2014, 2014.

[15] X. Yuan, E. B. Nuakoh, J. S. Beal, and H. Yu, "Retrieving relevant capec attack patterns for secure software development," in *Proceedings of the 9th Annual Cyber and Information Security Research Conference*. ACM, 2014, pp. 33–36.

[16] B. J. Berger, K. Sohr, and R. Koschke, "Automatically extracting threats from extended data flow diagrams," in *International Symposium on Engineering Secure Software and Systems*. Springer, 2016, pp. 56–71.

[17] L. Lambert, "Building security into your software development," Tech. Rep., 2018.

[18] S. S. Alqahtani, E. E. Eghan, and J. Rilling, "Sv-af—a security vulnerability analysis framework," in *Software Reliability Engineering (ISSRE), 2016 IEEE 27th International Symposium on*. IEEE, 2016, pp. 219–229.

[19] C. Wijesiriwardana and P. Wimalaratne, "Fostering real-time software analysis by leveraging heterogeneous and autonomous software repositories," *IEICE TRANSACTIONS on Information and Systems*, vol. 101, no. 11, pp. 2730–2743, 2018.

[20] M. Rath, M. Goman, and P. Mäder, "State of the art of traceability in open-source projects," 2017.

[21] C. Wijesiriwardana and P. Wimalaratne, "Software engineering data analytics: A framework based on a multi-layered abstraction mechanism," *IEICE Transactions on Information and Systems*, vol. 102, no. 3, pp. 637–639, 2019.

[22] E. Crifasi, S. Pike, Z. Stuedemann, S. M. Alnaeli, and Z. Altahat, "Cloud-based source code security and vulnerabilities analysis tool for c/c++ software systems," in *2018 IEEE International Conference on Electro/Information Technology (EIT)*. IEEE, 2018, pp. 0651–0654.

[23] R. Mahmood and Q. H. Mahmoud, “Evaluation of static analysis tools for finding vulnerabilities in java and c/c++ source code,” *arXiv preprint arXiv:1805.09040*, 2018.

[24] M. Abi-Antoun, D. Wang, and P. Torr, “Checking threat modeling data flow diagrams for implementation conformance and security,” in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. ACM, 2007, pp. 393–396.

[25] T. UcedaVelez and M. M. Morana, *Risk Centric Threat Modeling: Process for Attack Simulation and Threat Analysis*. John Wiley & Sons, 2015.

[26] D. Merritt, *Building expert systems in Prolog*. Springer Science & Business Media, 2012.

[27] A. Aamodt and E. Plaza, “Case-based reasoning: Foundational issues, methodological variations, and system approaches,” *AI communications*, vol. 7, no. 1, pp. 39–59, 1994.